Computability and complexity

ESISAR - INPG
Valence, France

http://membres-liglab.imag.fr/diazcaro/MA554.html

Alejandro Díaz-Caro
alejandro.diaz-caro@imag.fr

2010

Contents

Chapter 1

Reminder: Automata and grammars

1.1 Regular languages

Alphabet An *alphabet* Σ is a finite set of symbols.

Formal language A formal *language* over an alphabet Σ is a set of words, *i.e.* finite strings of symbols from Σ.

Regular languages The collection of *regular languages* over an alphabet Σ is defined recursively as follows:

- The empty language $\emptyset$ is a regular language.
- The empty string language $\{\varepsilon\}$ is a regular language.
- For each $a \in \Sigma$, the singleton language $\{a\}$ is a regular language.
- If A and B are regular languages, then $A \cup B$ (union), $A \cdot B$ (concatenation), A^* (Kleene star) and B^* are regular languages.
- No other languages over Σ are regular.

All finite languages are regular. Other typical examples include the language consisting of all strings over the alphabet $\{a, b\}$ which contain an even number of as, or the language consisting of all strings of the form: several as followed by several bs.

A simple example of a language that is not regular is the set of strings $\{a^n b^n \mid n \geq 0\}$.

1.2 Deterministic finite automata

1.2.1 Definitions

Deterministic finite automata A *deterministic finite automaton* is formally defined by the 5-tuple $\langle \mathcal{Q}, \Sigma, \delta, q_0, \mathcal{F} \rangle$, where:

- $\mathcal{Q}$ is a finite set of states.
- Σ is the alphabet of the automaton.
- δ is the transition function, that is, $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$.
- q_0 is the start state, that is, the state which the automaton is in when no input has been processed yet, where $q_0 \in \mathcal{Q}$.
- $\mathcal{F}$ is a set of states of $\mathcal{Q}$ (*i.e.* $\mathcal{F} \subseteq \mathcal{Q}$) called accepted states.

Input word An automaton reads a finite string of symbols $a_1a_2 \ldots a_n$, where $a_i \in \Sigma$, which is called a input word. The set of all words is denoted by Σ^*.

Run A run of the automaton on an input word $\omega = a_1a_2 \ldots a_n \in \Sigma^*$, is a sequence of states $q_0q_1q_2 \ldots q_n$, where $q_i \in \mathcal{Q}$ such that q_0 is a start state and $q_i = \delta(q_i - 1, a_i)$ for $0 < i \leq n$. In words, at first the automaton is at the start state q_0 and then automaton reads symbols of the input word in sequence. When automaton reads symbol a_i then it jumps to state $q_i = \delta(q_i - 1, a_i)$. q_n said to be the final state of the run.

Accepting word A word $\omega \in \Sigma^*$ is accepted by the automaton if $q_n \in \mathcal{F}$.

Recognized language An automaton can recognize a formal language. The recognized language $\mathcal{L} \subset \Sigma^*$ by an automaton is the set of all the words that are accepted by the automaton.

Recognizable languages The recognizable languages is the set of languages that are recognized by some automaton. For above definition of automata the recognizable languages are regular languages. For different definitions of automata, the recognizable languages are different.

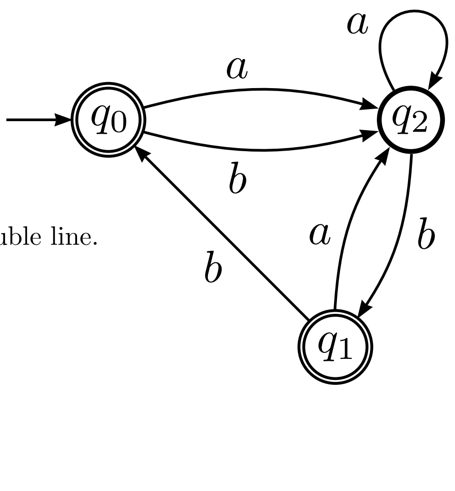
1.2.2 Example

The following 5-tuple is an automaton:

$$M = \langle \mathcal{Q} = \{q_0, q_1, q_2\}, \Sigma = \{a, b\}, \delta, q_0, \mathcal{F} = \{q_0, q_1\} \rangle$$

where δ is defined by
$$\begin{aligned} \delta(\_, a) &= q_2 \\ \delta(q_0, b) &= q_2 \\ \delta(q_1, b) &= q_0 \\ \delta(q_2, b) &= q_1 \end{aligned}$$

An easier way to define δ is graphically. As instance, the above automaton can be drawn as follows:



Where

- Each nodes represent a state.
- Final states are marked with a double line.

- The initial state is pointed by an arrow
- Transitions are represented by directed edges between states, labeled with the symbol of the alphabet which produces such a transition.

Let $\omega_1 = abbaab$ be an input word. The run of the automaton on such a word is the sequence $q_0q_2q_1q_0q_2q_2q_1$ which is an accepted word since q_1 is a final state.

Let $\omega_2 = bba$ be an input word. The run of the automaton on this word is the sequence $q_0q_2q_1q_2$ which is not accepted since q_2 is not a final state.

The recognized language $\mathcal{L}$ for this automaton can be characterized in the following way.

Let denote by s^* the string of zero or more occurrences of the symbol s, and by s^+ the string ss^* (*i.e.* one or more more occurrences of s).

Then we define $\mathcal{L}$ *recursively* by

1. $\varepsilon \in \mathcal{L}$
2. a^+b is in $\mathcal{L}$
3. ba^*b is in $\mathcal{L}$
4. If ω is in $\mathcal{L}$ then ωa^*b is in $\mathcal{L}$
5. $\mathcal{L}$ is the smallest set satisfying 1, 2, 3 and 4.

1.3 Nondeterministic finite automata

1.3.1 Definitions

Nondeterministic finite automata A *nondeterministic finite automaton* is a generalization of the finite automaton which may allows

- more than one transition corresponding to a symbol in a given state
- transitions on the empty symbol (noted by ε)
- transitions on words

Formal definition A *nondeterministic finite automaton* is formally defined by the 5-tuple $\langle \mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{F} \rangle$, where:

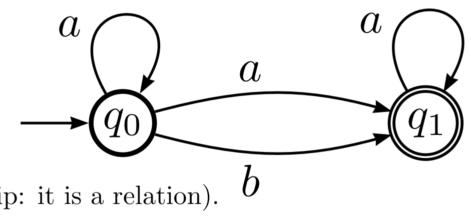
- $\mathcal{Q}$ is a finite set of states.
- Σ is the alphabet of the automaton.
- Δ is the transition relation, that is, $\Delta \subset \mathcal{Q} \times \Sigma^* \times \mathcal{Q}$
- q_0 is the start state, that is, the state which the automaton is in when no input has been processed yet, where $q_0 \in \mathcal{Q}$.
- $\mathcal{F}$ is a set of states of $\mathcal{Q}$ (*i.e.* $\mathcal{F} \subseteq \mathcal{Q}$) called accepted states.

1.3.2 Example

The following is a nondeterministic finite automaton:

$$N = \langle \mathcal{Q} = q_0, q_1, \Sigma = a, b, \Delta, q_0, \mathcal{F} = q_1 \rangle$$

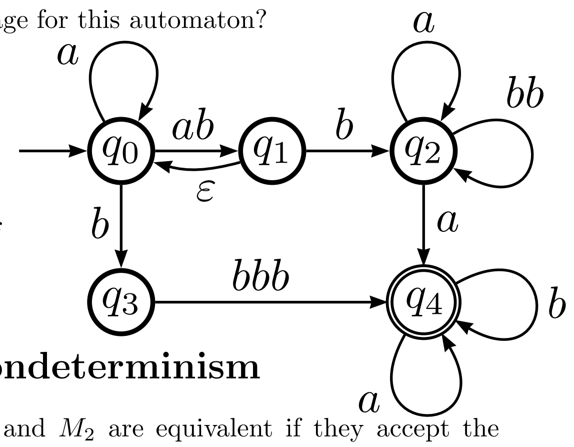
where Δ is implicitly defined by the following drawing



Exercises

1. Give the explicit definition of Δ. (Tip: it is a relation).
2. What is the recognized language for this automaton?

1.3.3 A second example



Exercise
Give the formal description of this nondeterministic automaton and its recognized language.

1.4 Removing the nondeterminism

Equivalence Two automata M_1 and M_2 are equivalent if they accept the same language, *i.e.* $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.

Theorem 1 *For all nondeterministic finite automaton, it is possible to construct an equivalent deterministic finite automaton.*

Exercise Show a deterministic finite automaton equivalent to the one at example 1.3.3

1.5 Regular grammars

Grammar A *grammar* is a 4-tupe $\langle N, \Sigma, S, R \rangle$ where:

- N is a finite set of nonterminal symbols.
- Σ is a finite set of terminal symbols.
- $S \in N$ is the start symbol.
- R is a set of *rewrite rules.* This rules are formed from a term at the left, an arrow ($\rightarrow$) and a term at the right. The terms at left and right can be any combination of symbols from N or Σ, provided that there is at least one symbol N on the left. The right side may be empty, which is indicated by ε. This is called ε-rule

Regular grammar A *regular grammar* is a grammar such that the rewriting rules are of one of the following forms:

- $B \to a$, where $B \in N$ and $a \in \Sigma$.
- $B \to aC$, where $B, C \in N$ and $a \in \Sigma$.
- $B \to \varepsilon$, where $B \in N$ and ε denotes the empty string.

Theorem 2 *Given a language Σ, every language generated by a regular grammar over Σ can be recognized by a finite automaton and every accepted languages by a finite automaton can be generated by a regular grammar.*

$$\{\mathcal{L}(G) : G \textit{ is a regular grammar over } \Sigma\} = \{\mathcal{L}(M) : M \textit{ is a FA over } \Sigma\}$$

Conversion Automaton / Grammar Let $M = \langle \mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{F}\rangle$ be a nondeterministic finite automaton. The regular grammar that generate the accepted language by M is

$$G = \langle N, \Sigma, S, R\rangle$$

where

- $N = \mathcal{Q}$
- $\Sigma = \Sigma$
- $S = q_0$
- $R = \{P \to xQ : (P, x, Q) \in \Delta\} \cup \{Q \to \varepsilon : Q \in \mathcal{F}\}$

Example The following regular grammar generates the accepted language from example 1.2.2:

Rewriting rules	Δ – relation
$A \to aC$	(q_0, a, q_2)
$B \to aC$	(q_1, a, q_2)
$C \to aC$	(q_2, a, q_2)
$A \to bC$	(q_0, b, q_2)
$B \to bA$	(q_1, b, q_0)
$C \to bB$	(q_2, b, q_1)
$A \to \varepsilon$	$q_0 \in \mathcal{F}$
$B \to \varepsilon$	$q_1 \in \mathcal{F}$

Notice that we used A, B, C instead of q_0, q_1, q_2 to improve the reading.

Conversion Grammar / Automaton Let $G = \langle N, \Sigma, S, R\rangle$ be a regular grammar. It is always possible to define a grammar $G' = \langle N', \Sigma, S, R'\rangle$ that generates the same language but without having rules with the right side being only one nonterminal symbol.

We define a nondeterministic automaton $M = \langle \mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{F}\rangle$ where

- $\mathcal{Q} = N'$
- $\Sigma = \Sigma$

- $q_0 = S$
- $\Delta = \{(P, x, Q) : \text{there is a rule of the shape } P \to xQ \text{ in } R'\}$
- $\mathcal{F} = \{X : X \to \varepsilon \text{ is a rule in } R'\}$

Example Let G be the following grammar:

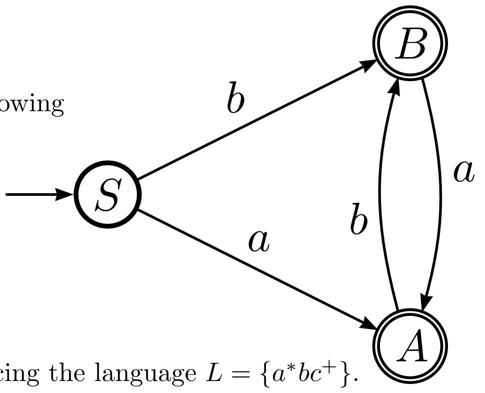
$$\begin{array}{c} S \to aA \\ A \to bB \\ B \to aA \\ S \to bB \\ A \to \varepsilon \\ B \to \varepsilon \end{array}$$

This grammar has no rules with the right side being only one nonterminal symbol, so we do not need to make any modification.

Begin $N = \{S, A, B\}$ and $\Sigma = \{a, b\}$, we find the following automaton $M = \langle \mathcal{Q}, \Sigma, \Gamma, q_0, \mathcal{F}\rangle$ where

$$\begin{array}{rcl} \mathcal{Q} & = & \{A, B, S\} \\ \Sigma & = & \{a, b\} \\ \Delta & = & \{(S, a, A), (S, b, B), (A, b, B), (B, a, A)\} \\ q_0 & = & S \\ \mathcal{F} & = & \{A, B\} \end{array}$$

The transition diagram is the following



1.5.1 Exercises

1. Give a regular grammar producing the language $L = \{a^*bc^+\}$.
2. Give a deterministic finite automaton accepting the same language.

1.6 Pushdown automata

Intuition A pushdown automaton is a finite automaton that can make use of a stack containing data. Transitions are denoted by a 5-tuple. The intuition is that $(a, b, c; d, e)$ means "Being on state a, read b from the entrance and take c from the stack, then move to state d and write e in the stack".

Formalization A *pushdown automaton* is a 6-tuple $\langle \mathcal{Q}, \Sigma, \Gamma, T, q_0, \mathcal{F}\rangle$ where

- $\mathcal{Q}$ is a finite set of states.
- Σ is a finite set of symbols which is called the input alphabet.
- Γ is a finite set of symbols which is called the stack alphabet.

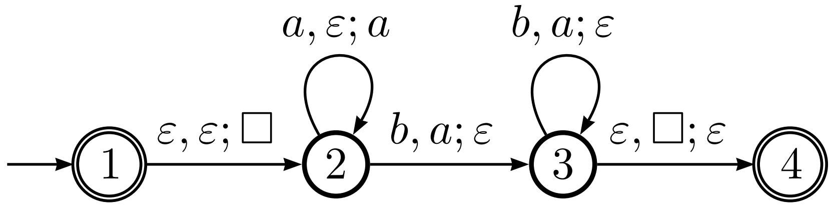
- T is a finite set of transitions. $T \subseteq \mathcal{Q} \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times \mathcal{Q} \times (\Gamma \cup \{\varepsilon\})$.
- $q_0 \in \mathcal{Q}$ is the start state.
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of final states.

Example Let M be the following pushdown automaton:

$$M = \langle \{1,2,3,4\}, \{a,b\}, \{a,b,\square\}, T, 1, \{1,4\} \rangle$$

where $T = \{(1,\varepsilon,\varepsilon;2,\square),(2,a,\varepsilon;2,a),(2,b,a;3,\varepsilon),(3,b,a;3,\varepsilon),(3,\varepsilon,\square;4,\varepsilon)\}$.

Here it is the transitions diagram


$a,\varepsilon;a$
$b,a;\varepsilon$
$\varepsilon,\varepsilon;\square$
$b,a;\varepsilon$
$\varepsilon,\square;\varepsilon$
1
2
3
4

Exercise Show that this automaton accept the language $\{a^n b^n : n \geq 1\}$

1.7 Context-free grammars

Definition A *context-free grammar* is a grammar such that each rewrite rule has only one nonterminal symbol on the left of the arrow.

Example 1 The canonical example of a context free grammar is parenthesis matching, which is representative of the general case. Let $N = \{S\}$ and $\Sigma = \{(,)\}$. The rewriting rules are $S \to SS$
$S \to (S)$
$S \to ()$

The first rule allows Ss to multiply; the second rule allows Ss to become enclosed by matching parentheses; and the third rule terminates the recursion.

Starting with S, and applying the rules, one can construct:

$$S \to SS \to SSS \to (S)SS \to ((S))SS \to ((SS))S(S)$$
$$\to ((()S))S(S) \to ((()()))S(S) \to ((()()))()(S) \to ((()()))()(())$$

Example 2 A second canonical example is two different kinds of matching nested parentheses. Let $N = \{S\}$ and $\Sigma = \{(,),[,]\}$. The rewriting rules are
$S \to SS$
$S \to ()$
$S \to (S)$
$S \to []$
$S \to [S]$

The following sequence can be derived in that grammar: ([[[()()[][]]]([])])

Exercise Show a context-free grammar generating the language $\{a^m b^n c^m : m \geq 0, n \geq 1\}$.

Theorem 3 *For all context-free grammar G, there exists a pushdown automaton M such that $\mathcal{L}(G) = \mathcal{L}(M)$.*

Theorem 4 *For all pushdown automaton M, there exists a context-free grammar G such that $\mathcal{L}(G) = \mathcal{L}(M)$.*

Chapter 2

Turing machines

Alan Mathison Turing was born in London, 23 June 1912. He gave a definition of computation and an absolute limitation on what computation could achieve. Turing's most important work:
Turing, A. M., Computing machinery and intelligence*, Mind 50:433–460.*

Introduction Since the family of regular languages is a subclass of the family of context-free languages, we say pushdown automata are more powerful than finite automata. However, there are still languages that are not context-free, for example, $\{a^n b^n c^n\}$ or $\{ww \mid w \in \{a, b\}^*\}$.

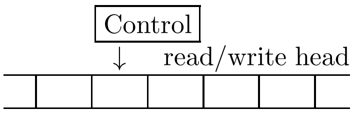
The main difference between a pushdown automaton and a finite automaton is the storage. A finite automaton has no storage. A pushdown automaton has a stack.

We might get even more powerful automata if we add a more flexible storage method. For example, what may happen if a pushdown automaton is equipped with 2 (instead of 1) stacks? 3 stacks? a queue?

This approach leads to the concept of a Turing machine, which, in turn, leads to a precise definition of mechanic or algorithmic computation.

2.1 The Standard Turing Machine

Informal description The storage of a *Turing machine* is considered as a single 1-dimensional array which extends indefinitely in both directions. The storage can hold an unlimited amount of information. Information can be written to and read from the storage in any order. This storage is called a *tape*.



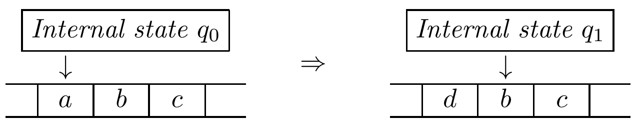
The tape is divided into *cells*. Each cell is capable of holding one *symbol* of a given alphabet. There is a *read/write head* that can travel on the tape to the left or to the right and can read or write a symbol on each move. There is no input nor output device for a Turing machine. Instead, input and output are done on the machine's tape.

Formal definition A *Turing machine* is 7-tuple $\langle \mathcal{Q}, \Sigma, \Gamma, \delta, q_0, \Box, \mathcal{F} \rangle$, where:

- $\mathcal{Q}$ is the finite set of internal states.

- $\Sigma \subseteq \Gamma \setminus \{\Box\}$ is the input alphabet (which is a finite set of symbols).
- Γ is the tape alphabet (which is a finite set of symbols).
- δ is the transition function, that is, $\delta : \mathcal{Q} \times \Gamma \to \mathcal{Q} \times \Gamma \times \{L, R\}$
- $q_0 \in \mathcal{Q}$ is the initial state.
- $\Box \in \Gamma$ is the *blank* tape symbol.
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of final states.

Example 1 *Assume $\delta(q_0, a) = (q_1, d, R)$. The transition would be*



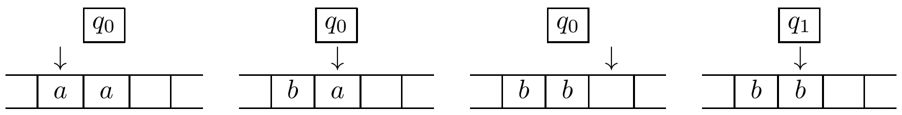
A Turing machine can be thought of as a simple computer with a processing unit (which has a finite memory) and a tape (which has unlimited capacity).

It has an internal state. It can read the symbol in the cell just under the r/w head. Based on the internal state and the symbol just read, it consults the transition function δ, writes a symbol on the cell and moves to the left or to the right.

The transition function defines the behavior of a Turing machine. It is comparable to the program in a conventional computer. The Turing machine starts from the initial state with some symbols on the tape. Eventually, the Turing machine may enter into a halt state. A Turing machine is said to halt whenever it reaches a configuration for which δ is not defined. Note that δ is a partial function. We usually assume that no transitions are defined for any final state so that the Turing machine will halt whenever it enters a final state.

Example 2 *Consider the following δ transition function:*

$$\begin{aligned} \delta(q_0, a) &= (q_0, b, R) \\ \delta(q_0, b) &= (q_0, b, R) \\ \delta(q_0, \Box) &= (q_1, \Box, L) \end{aligned}$$



This Turing machine replaces any a in the tape by a b, and halt as soon as it reaches a blank symbol.

Example 3 *Consider the following δ transition function:*

$$\begin{aligned} \delta(q_0, a) &= (q_1, a, R) \\ \delta(q_0, b) &= (q_1, b, R) \\ \delta(q_0, \Box) &= (q_1, \Box, R) \\ \delta(q_1, a) &= (q_0, a, L) \\ \delta(q_1, b) &= (q_0, b, L) \\ \delta(q_1, \Box) &= (q0, \Box, L) \end{aligned}$$

Assume the input is ab. This Turing machine will move left and right alternatively, without changing the tape. Actually this behavior will always happen whatever is on the tape. We say the Turing machine enters an infinite loop. It will never halt.

The following table is a representation of the transition function in the last example:

δ	a	b	$\Box$
q_0	(q_1, a, R)	(q_1, b, R)	$(q_1, \Box, R)$
q_1	(q_0, a, L)	(q_0, b, L)	$(q_0, \Box, L)$

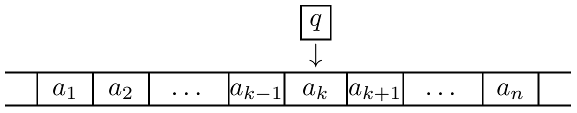
There are variations of the Turing machines. However, we will summarize the model of a standard Turing machine:

- The tape is unbounded in both directions. This means that the read/write head can go left and right without any restriction.
- A Turing machine is *deterministic* in that δ allows at most one move in any configuration.
- There is no special input file. We assume the initial tape contents constitute the input. Similarly, there is no special output file. When a Turing machine halts, (part of) the tape contents constitute the output.

The *configuration* (or *instantaneous description*, a situation at a particular instant) of a Turing machine consists in the current state of the control unit, the contents of the tape, and the position of the r/w head. We use the notation

$$a_1a_2\dots a_{k-1}\boxed{q}a_ka_{k+1}\dots a_n$$

to denote a configuration, where q is the current state, a_k is where the r/w head is, and $a_1a_2\dots a_n$ is the contents of the tape. The unspecified part of the tape is assumed to contain blanks.



We use the notation $bbq_0ab \vdash bbbq_0b$ (for example 2) to denote a 1-step transition. The relation $\vdash^*$ is the reflexive and transitive closure of $\vdash$.

For the example 2 we may have the following transitions:

$$q_0aa \vdash bq_0a \vdash bbq_0\Box \vdash bq_1b$$

or

$$q_0aa \vdash^* bq_1b$$

Lemma 1 $a_1a_2\dots a_{k-1}pba_{k+1}\dots a_n \vdash a_1a_2\dots a_{k-1}cqa_{k+1}\dots a_n$ *if and only if* $\delta(p,b) = (q,c,R)$.

Similarly, $a_1a_2\dots a_{k-1}pba_{k+1}\dots a_n \vdash a_1a_2\dots qa_{k-1}ca_{k+1}\dots a_n$ *if and only if* $\delta(p,b) = (q,c,L)$.

This lemma should be considered as the definition of $\vdash$.

A Turing machine *halts* when it enters a configuration, say

$$a_1 a_2 \dots a_{k-1} q a_k a_{k+1} \dots a_n$$

in which $\delta(q, a_k)$ is not defined. This configuration is called a halting configuration. The sequence of configurations leading from a given initial configuration to a halting configuration is called a computation.

Example 3 shows that a Turing machine may not always halt. We use the notation $xqy \vdash^* \infty$ to denote a non-halting computation.

A Turing machine can serve as a *language acceptor* in the sense that, assuming the initial tape contents is the string to be tested, the rest of the tape contains all blanks, the control unit is in the initial state, and the r/w head is placed at the leftmost non-blank symbol (that is the first symbol of the string), if the Turing machine eventually halts at a final state, then we say the Turing machine accepts the string.

Definition Let $M = \langle \mathcal{Q}, \Sigma, \Gamma, \delta, q_0, \Box, \mathcal{F} \rangle$ be a Turing machine. The language accepted by M is

$$L_M = \{ w \in \Sigma^+ \mid q_0 w \vdash^* xpy \text{ for some } p \in \mathcal{F}, x, y \in \Gamma^* \}$$

Note that we explicitly require that $\Box \notin \Sigma$ so that we know where the input starts and ends. Without this restriction, we can never be sure that there is any more input symbols in cells that are not examined by the machine.

Notice that $\varepsilon \notin L_M$.

On the other hand, if a string $w \notin L_M$, either the Turing machine M halts on a non-final state; or the Turing machine M may never halt.

This raises a problem: how can we decide if M will halt on an input string.

Example 4 *Let $\Sigma = \{0, 1\}$. Desing a Turing machine that accepts the regular language 00^*.*
Solution:

$$\begin{aligned} \delta(p, 0) &= (p, 0, R) \\ \delta(p, \Box) &= (q, \Box, R) \end{aligned}$$

This Turing machine starts and stays in state p when reading 0's. At the end (that is, encountering the first $\Box$), it switches to state q, which is the only final state. Whenever the Turing machine meets a 1, it hangs.

$$p000 \vdash 0p00 \vdash 00p0 \vdash 000p\Box \vdash 000\Box q$$

$$p001 \vdash 0p01 \vdash 00p1$$

This machine also enters in state q if the tape contains only $\Box$. We may interpret that it accepts ε. However, according to a previous definition, a Turing machine never accepts ε.

Example 5 *Let $\Sigma = \{a, b\}$. Design a Turing machine that accepts the regular language $L = \{a^n b^n \mid n \geq 1\}$.*
Solution: This Turing machine starts and stays in state p when reading a's. It changes a to x during this left-to-right scan. When it encounters the first b, it changes the b to y and moves from right to left to find the last x. This last x is changed to y and the machine again moves from left to right to locate the next b. After all b's are examined, it changes to state s and moves from right to left to the blank just before the input. At this time, the machine enters state t, which is the only final state.

$$\begin{aligned}
\delta(p, a) &= (p, x, R) \\
\delta(p, \Box) &= \textit{fail – no more b's at all} \\
\delta(p, b) &= (q, y, L) \\
\delta(q, y) &= (q, y, L) \\
\delta(q, x) &= (r, y, R) \\
\delta(q, \Box) &= \textit{fail — more b's than a's} \\
\delta(r, y) &= (r, y, R) \\
\delta(r, b) &= (q, y, L) \\
\delta(r, \Box) &= (s, \Box, L) \\
\delta(s, y) &= (s, y, L) \\
\delta(s, \Box) &= (t, \Box, R) \\
\delta(s, x) &= \textit{fail — more a's than b's}
\end{aligned}$$

$$\begin{aligned}
paabb &\vdash xpabb \vdash xxpbb \vdash xqxyb \vdash xyryb \vdash xyyrb \vdash xyqyy \vdash xqyyy \vdash qxyyy \\
&\vdash yryyy \vdash yyryy \vdash yyyry \vdash yyyyr \vdash yyysy \vdash yysyy \vdash ysyyy \vdash syyyy \\
&\vdash s\Box yyyy \vdash tyyyy
\end{aligned}$$

Another possible definition could be

$$\begin{aligned}
\delta(p, a) &= (q, x, R) \\
\delta(q, a) &= (q, a, R) \\
\delta(q, y) &= (q, y, R) \\
\delta(q, b) &= (r, y, L) \\
\delta(r, y) &= (r, y, L) \\
\delta(r, a) &= (r, a, L) \\
\delta(r, x) &= (p, x, R) \\
\delta(p, y) &= (s, y, R) \\
\delta(s, y) &= (s, y, R) \\
\delta(s, \Box) &= (t, \Box, R)
\end{aligned}$$

Notice that the difference between one and the other is how it pairs the a's with the b's. The first one pairs the first a with the last b, the second a with the b after the last one, and so on. The second TM pairs the first a with the first b and so on. In both cases, if the input string is not in the language, the machine will halt in a non-final state.

Example 6 *Design a Turing machine accepting the language*

$$L = \{a^n b^n c^n \mid n \geq 1\}$$

Note that, though $\mathcal{L}$ is not context-free, there is a Turing machine accepting $\mathcal{L}$.
Solution:

$$\begin{array}{rclrcl}
\delta(p,a) &=& (q,x,R); & \delta(q,a) &=& (q,a,R); \\
\delta(q,y) &=& (q,y,R); & \delta(q,b) &=& (r,y,R); \\
\delta(r,z) &=& (r,z,R); & \delta(r,c) &=& (s,z,L); \\
\delta(s,z) &=& (s,z,L); & \delta(s,b) &=& (s,b,L); \\
\delta(s,y) &=& (s,y,L); & \delta(s,a) &=& (s,a,L); \\
\delta(s,x) &=& (p,x,R); & & & \\
\delta(p,y) &=& (t,y,R); & \delta(t,y) &=& (t,y,R); \\
\delta(t,z) &=& (t,z,R); & \delta(t,\Box) &=& (u,\Box,R).
\end{array}$$

This example shows that Turing machines can accept some languages that are not context-free. This indicates that Turing machines are more powerful than pushdown automata (in terms of the languages that they can recognized).

Note that this TM makes use of only a very limited part of the tape. This kind of TMs is called linear-bounded automata.

2.1.1 Turing machines as transducers

The primary purpose of a digital computer is to transform input to output. It acts as a *transducer*. Since we want to use Turing machines as a model of digital computers, we have to view a Turing machine as a transducer as well.

The input of a computation would be all the non-blank symbols initially on the tape. At the conclusion of a computation, the output will be whatever is left on the tape. Thus, a Turing machine may be considered as a function, defined by

$$\hat{w} = f(w)$$

provided that

$$q_0 w \vdash_M^* q_f \hat{w}$$

where q_f is a final state.

Definition A function f with a domain D is said to be *Turing-computable* (or *computable*) if there is a Turing machine M such that, for all $w \in D$, $q_0 w \vdash_M^* q_f f(w)$, where q_f is a final state.

This seemingly naive definition is very important. All the common mathematical functions are computable in this sense.

Example 7 *Design a Turing machine that computes $x + y$, where x and y are natural numbers.*
Solution: Let $\Gamma = \{0, 1\}$. We adopt the convention that a natural number, say n is represented by n 1's on the tape. We use the notation $\overline{n}$ to denote n consecutive 1's.

We will assume that x and y are placed on the tape consecutively, separated by a 0. Initially, the r/w head is located at the leftmost symbol of x.

We want to design a Turing machine such that $q_0\overline{x}0\overline{y} \vdash^* q_f\overline{(x+y)}0$.

This Turing machine simply moves the 0 *to the right end.*

$$\begin{aligned}
\delta(p,1) &= (p,1,R)\\
\delta(p,0) &= (q,1,R)\\
\delta(q,1) &= (q,1,R)\\
\delta(q,\Box) &= (r,\Box,L)\\
\delta(r,1) &= (s,0,L)\\
\delta(s,1) &= (s,1,L)\\
\delta(s,\Box) &= (t,\Box,R)
\end{aligned}$$

$p110111 \vdash 1p10111 \vdash 11p0111 \vdash 111q111 \vdash 1111q11 \vdash 11111q1 \vdash 111111q\Box \vdash 11111r1 \vdash 1111s10 \vdash 111s110 \vdash 11s1110 \vdash 1s11110 \vdash s111110 \vdash s\Box111110 \vdash t111110$

Unary notation results in very simple Turing machines.

Example 8 *Design a Turing machine that copies strings of 1's. That is, let* $w \in \{1\}^*$ *. We want the following computation:* $q_0w \vdash^* q_fww$*.*

This machine actually computes $2 \cdot x$*, for a positive integer* x*.*
Solution: Our program consists of 4 steps:

1. *Replace every* 1 *with* x *in a left-to-right scan.*
2. *Find the rightmost* x *in a right-to-left scan and replace it with 1.*
3. *Add at the right end a symbol 1.*
4. *Repeat steps 2 and 3 until there are no more* x*'s.*

In the following machine, p is the initial state and s is the (only) final state.

$$\begin{aligned}
\delta(p,1) &= (p,x,R)\\
\delta(p,\Box) &= (q,\Box,L)\\
\delta(q,1) &= (q,1,L)\\
\delta(q,x) &= (r,1,R)\\
\delta(q,\Box) &= (s,\Box,R)\\
\delta(r,1) &= (r,1,R)\\
\delta(r,\Box) &= (q,1,L)
\end{aligned}$$

$p11 \vdash xp1 \vdash xxp\Box \vdash xqx \vdash x1r\Box \vdash xq11 \vdash qx11 \vdash 1r11 \vdash 11r1 \vdash 111r\Box \vdash 11q11 \vdash 1q111 \vdash q1111 \vdash q\Box1111 \vdash s1111\Box$

Example 9 *Design a Turing machine that copies strings of 1's. That is, let* $w \in \{1\}^*$ *. We want the following computation:* $q_0w \vdash^* q_fwww$*.*

This machine actually computes $3 \cdot x$*, for a positive integer* x*.*
Solution: Our program consists of 4 steps:

1. *Replace every* 1 *with* x *in a left-to-right scan.*
2. *Find the rightmost* x *in a right-to-left scan and replace it with 1.*

3. Add at the right end two symbols 11.

4. Repeat steps 2 and 3 until there are no more x's.

In the following machine, p is the initial state and s is the (only) final state.

$$\begin{array}{rcl}
\delta(p,1) & = & (p,x,R)\\
\delta(p,\square) & = & (q,\square,L)\\
\delta(q,1) & = & (q,1,L)\\
\delta(q,x) & = & (r,1,R)\\
\delta(q,\square) & = & (s,\square,R)\\
\delta(r,1) & = & (r,1,R)\\
\delta(r,\square) & = & (t,1,R) \ \textit{new move}\\
\delta(t,\square) & = & (q,1,L) \ \textit{new move}
\end{array}$$

$p11 \vdash xp1 \vdash xxp\square \vdash xqx \vdash x1r\square \vdash x11t\square \vdash x11q1 \vdash x1q11 \vdash xq111 \vdash qx111 \vdash 1r111 \vdash 11r11 \vdash 111r1 \vdash 1111r\square \vdash 11111t\square \vdash 11111q1 \vdash 1111q11 \vdash 111q111 \vdash 11q1111 \vdash 1q11111 \vdash q111111 \vdash q\square111111 \vdash s111111\square$

Example 10 *Let x and y be two natural numbers. Write a Turing machine that compares them. If $x \geq y$, the machine should halt in a final state. Otherwise, it should halt in a non-final state. Specifically, let $\overline{x}$ and $\overline{y}$ be the unary representations of x and y, respectively. We want*

$$\begin{array}{ll}
p\overline{x}0\overline{y} \vdash^* g\overline{x}0\overline{y} & \textit{if } x \geq y\\
p\overline{x}0\overline{y} \vdash^* l\overline{x}0\overline{y} & \textit{if } x < y
\end{array}$$

Solution: We match each digit of $\overline{x}$ with a digit of $\overline{y}$ and see which is exhausted first. This example shows that a Turing machine can make decisions based on arithmetical comparisons, which occur frequently in computer programming.

$$\begin{array}{lll}
\delta(p,1){=}(q,x,R) & \delta(q,1){=}(q,1,R) & \delta(q,0){=}(s,0,R)\\
\delta(s,y){=}(s,y,R) & \delta(s,1){=}(t,y,L) & \delta(s,\square){=}(h,\square,L) \text{ i.e. } x \geq y\\
\delta(t,y){=}(t,y,L) & \delta(t,0){=}(t,0,L) & \delta(t,1){=}(t,1,L)\\
\delta(t,x){=}(p,x,L) & & \\
\delta(p,0){=}(u,0,R) & \delta(u,y){=}(u,y,R) & \delta(u,\square){=}(h,\square,L) \text{ i.e. } x = y\\
\delta(u,1){=}(m,1,L) & \text{i.e. } x < y & \\
\delta(h,y){=}(h,1,L) & \delta(h,0){=}(h,0,L) & \delta(h,1){=}(h,1,L)\\
\delta(h,x){=}(h,1,L) & \delta(h,\square){=}(g,\square,R) & \\
\delta(m,y){=}(m,1,L) & \delta(m,0){=}(m,0,L) & \delta(m,1){=}(m,1,L)\\
\delta(m,x){=}(m,1,L) & \delta(m,\square){=}(l,\square,R) &
\end{array}$$

Exercise In the previous example, there is a mistake. Find it and correct it.

Example 11 *This TM computes the modulo function*

$$f(x) = x \bmod 3$$

We assume that the input x is expressed in unary notation and the r/w head is placed at the leftmost 1 of x.

$$\begin{array}{lll}
\delta(p_0,1){=}(p_1,x,R) & \delta(p_1,1){=}(p_2,x,R) & \delta(p_2,1){=}(p_3,x,R)\\
\delta(p_3,1){=}(p_1,x,R) & \delta(p_3,\square){=}(p_4,\square,L) & \delta(p_4,1){=}(p_4,1,L)
\end{array}$$

An operation, such as addition or multiplication, is a function, which, in turn, is a special relation. A relation is a set of tuples. A tuple can be encoded as a string. Thus, an operation may be viewed as a set of strings or a language. A TM accepting a language is said to be able to perform the corresponding operation.

2.2 Turing's Thesis

Turing machines are conceptually simple. In reality, it is quite tedious to construct (and prove) a Turing machine for a non-trivial problem, say sorting.

Turing's Thesis *Any computation that can be done by* mechanical means *can be done by a Turing machine.*

This is *not* a theorem since we do not have a rigorous definition of a mechanical means. Though we can define such a computation model $\mathcal{M}$ (remember Turing machines are also a model of computation), the most we can do is to prove that Turing machines are equal to $\mathcal{M}$.

Rather, Turing machines can be viewed as a *definition* of mechanical computation. Other computation models can, thus, be compared to Turing machines.

It is logically possible to find a more powerful model. However, all attempts to find a more powerful model have failed up to now.

Though anybody can propose a new model and a similar thesis. Turing's thesis is unusual in that it agrees with our current experience and observations in the study of computation, which include

- Anything that could be done with existing digital computers can be done with Turing machines.
- Nobody has not yet discovered a problem that can be solved with an algorithm (in the intuitive sense) but that cannot be solved with a Turing machine.
- All the proposed computation models up to now are equivalent to Turing machines.

What Turing machines are to computer science is what Newton's laws of motion are to classical physics. Newton's laws are not logical necessity; rather they are *plausible* models that can explain (and predict) much of the physical world.

We may use *algorithms* as synonyms for *Turing machines.*

Definition An *algorithm* for a function $f : D \to R$ is a Turing machine, M, which, when given any input $d \in D$ on its tape, eventually halts with the correct answer $f(d) \in R$ on its tape. Specifically, we require that, for all $d \in D$, $q_0 d \vdash_M^* q_f f(d)$, with $q_f \in \mathcal{F}$.

Once we have asserted that whatever a C program can do can be done with a Turing machine (that is, Turing's thesis), we could use C, instead of Turing machines, to discuss algorithms because Turing machines are tedious to use.

2.3 Multi-tape Turing machine

Equivalence of multi-tape TMs with normal TMs

Theorem 5 *Multi-tape Turing machines are as powerful as regular Turing machines.*

Definition A multi-tape Turing machine has n tapes ($n \geq 1$), a control, and a head for each tape. In the initial configuration, the first tape contains the input and its head starts at beginning of the input. The rest of the tapes are blank.

Idea of proof Given a multi-tape TM, M, produce a single-tape TM, M', such that $\mathcal{L}(M) = \mathcal{L}(M')$.

To simulate M with M' what we do basically is to use a new symbol, $\sharp$, to separate the contents of the different tapes (note that the last symbol of M' is $\sharp$). Then for each of the simulated tapes we are going to put a dot above the tape symbol where the head is. So the alphabet for M' is

$$\Gamma_{M'} = \Gamma_M \cup \{\dot{s} \mid s \in M\} \cup \{\sharp\}$$

The following is an example of what we do.

Say we want simulate writing a symbol σ, then moving to the right of where the head on tape n is:

- Move the head to the most-left position (the beginning of the input).
- Move the head to the $(n-1)$-th $\sharp$ (Say that the first tape is the $n = 1$ tape. So if the head of M is on the first tape, we simply stay at the first symbol of M')
- Move the head over the first dotted symbol.
- Replace that symbol with σ and move the head to the right.
- If the head is over a non-$\sharp$ symbol, then put a dot above the current symbol and we are done. Otherwise, if the head is over a $\sharp$ symbol, that means we want to move to a previously blank symbol. We can accomplish this by, first replacing the $\sharp$ we are on with a blank with a dot over it, then shift everything over to the right by one.

2.4 Non-deterministic Turing machines

Definition Similar to other forms of non-deterministic computations, the computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. If some branch of the computation leads to the accept state, the machine accepts its inputs. Without loss of generality, assume the tree is binary.

The root of the tree would be the start configuration and each node is a possible continuation from the root node. Note that we should use a breadth-first, rather than a depth-first, search to explore the tree. If we did the latter, we could easily be tracing an infinite branch while missing the accepting configurations of some other branches.

Theorem 6 *Non-deterministic TMs are no more powerful than regular TMs.*

General idea of the proof Using a multi-tape TM, add a new tape to represent each 'branch' of the non-deterministc TM. So to branch, add a new tape and copy the tape that is branching to the new tape. Then take, say, the first continuation on the original tape, and the second continuation on the new tape. Keep in mind that we should use a breadth-first search, so we do only one computation on each of the node-representing tape and do at most one branching before we go back to computing the first node-representing tape.

2.5 Universal Turing machine

Let $\langle M \rangle$ denote the string representing a Turing machine M[1]. Furthermore, let $\langle M, \omega \rangle$ denote the string representing M and its input, ω.

Let $A_{TM} = \{\langle M, \omega \rangle \mid M \text{ is a TM that accepts } \omega\}$.

Theorem 7 *A_{TM} is recognizable.*

This is same as saying that there is a Turing machine U ("Universal Turing-machine"), that accepts A_{TM}.

Program Trivally assume that the format is correct.

1. Run M on input ω.

2. If M accepts, accept; if M rejects, reject.

How? Let's say U has 3 tapes. 1st tape for the tape of the simulated machine (SM), 2nd tape for SM's description. 3rd tape for scratch-work, to keep track of what state we are in.

1. Copy SM to tape 2, then delete SM from tape 1.

2. Copy q_0 to tape 3.

3. Find a rule in δ that starts with the state we are in.

4. Find a rule that works on the symbol we are over.

5. Copy the new state in that rule onto tape 3.

6. Write the symbol in that rule onto tape 1 over the old symbol.

7. Move Left/Right as appropriate, by the length of a symbol on tape 1.

8. If the state on tape 3 is either q_{accept} or q_{reject}, then we accept or reject, otherwise go back to step 3.

[1]For example, $\langle M \rangle$'s alphabet can be $\Gamma_M \cup \mathcal{Q}_M \cup \{L, R, \rightarrow, ,, \sharp\}$, so we could represent the transition function by $state, symbol \rightarrow state, symbol, move$ and separate each entrance by $\sharp$.

2.6 The halting problem

Definition A *decider* is a Turing machine which always explicitly accepts or rejects (always terminates). A language is said to be *decidable* if it is recognized by a decider.

The halting problem is a *decision* problem which can be stated as follows: Given a description of a Turing machine and an input, decide whether the machine will eventually halt when run with that input, or will run forever.

Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible $\langle M, \omega \rangle$ pairs *cannot exist.*

Theorem 8 (Halting problem) *A_{TM} is undecidable.*

Suppose there exists a Turing machine D which can decide whether an input consisting in a pair TM-input halts or not. Then, using D, we can decide $\mathcal{L} = \{\langle M \rangle \mid M \text{ accepts } \langle M \rangle\}$, and also we can build a D^* which can decide $\bar{\mathcal{L}} = \{\langle M \rangle \mid M \text{ does not accept } \langle M \rangle\}$.

Question: What does D^* on input $\langle D^* \rangle$? Given our assumption that D^* is a decider, there are two possible cases:

1. Accept. Then $\langle D^* \rangle \in \bar{\mathcal{L}} \Rightarrow D^*$ is a TM that does not accepts $\langle D^* \rangle$.
2. Reject. Then $\langle D^* \rangle \in \mathcal{L} \Rightarrow D^*$ is a TM that accepts $\langle D^* \rangle$.

For case 1, we accept because D^* rejects $\langle D^* \rangle$, however, by accepting we are also saying that D^* accepts $\langle D^* \rangle$; similarly for case 2, we reject because D^* accepts $\langle D^* \rangle$, however, by rejecting we are also saying that D^* rejects $\langle D^* \rangle$. In both cases we get a contradiction, since they both accept and reject their inputs. Thus, our assumption that there exists such a D must incorrect, and therefore the halting problem is undecidable.

The Library of Congress example This is an enlightening, simple example. Say we want to write two books, the first one listing all the books which refer to themselves, and the second one listing all the books which does not refer to themselves.

We run into trouble for the second one: If it includes itself in the list, it would contradict the condition that it only contains books that do not refer themselves, but if it does not includes to itself, then it would be incomplete since it is supposed to contain all books that do not refer to themselves.

2.7 Unrecognizable Languages for Turing Machines

Just as FAs, NDFAs, and PDAs each had some associated class of language that they could not "recognize", so do TMs. We call these languages *Turing-unrecognizable.*

We introduce two theorems which we will use in our treatment of unrecognizable and undecidable languages.

Theorem 9 *$\overline{A_{TM}}$ is Turing-unrecognizable.*

Theorem 10 *$\mathcal{L}$ is decidable $\iff$ $\mathcal{L}$ and $\bar{\mathcal{L}}$ are both recognizable.*

Note that Theorem 10 is the stronger statement. In fact, Theorem 10 implies Theorem 9.

As usual, in order to prove Theorem 10 (the stronger of the two statements) we must prove both directions. We will start with the simpler direction.

1. $\mathcal{L}$ is decidable $\Rightarrow$ $\mathcal{L}$ and $\bar{\mathcal{L}}$ are recognizable.

   (a) $\mathcal{L}$ is decidable $\Rightarrow$ $\mathcal{L}$ is recognizable. This is the trivial case. The same TM that decides $\mathcal{L}$ also recognizes it.

   (b) $\mathcal{L}$ is decidable $\Rightarrow$ $\bar{\mathcal{L}}$ is recognizable. To show this is the case, simply flip the results derived when deciding $\mathcal{L}$. So when $\mathcal{L}$ accepts, $\bar{\mathcal{L}}$ rejects, and vice versa.

2. $\mathcal{L}$ and $\bar{\mathcal{L}}$ are recognizable $\Rightarrow$ $\mathcal{L}$ is decidable.

   Imagine you have a couple of machines M and M' such that M recognizes $\mathcal{L}$ and M' recognizes $\bar{\mathcal{L}}$. Keep in mind that each machine will either accept, reject, or loop forever on input. Because the two languages are complements of one another, however, they can never both accept or both reject the same input. This idea leads us to our proof:

   On input ω, simultaneously simulate M on input ω, and M' on the same input, ω. If M accepts, accept. If M' accepts, reject.

   Clearly then, $\mathcal{L}_D = \mathcal{L}_M = \mathcal{L}$ and D, being our decider, always halts.

   To sum up, if w is a string then either $w \in \mathcal{L}$ or $w \notin \mathcal{L}$. In the former case, M will eventually accept w. In the latter case, M' will eventually accept w. Thus, D will halt eventually.

2.7.1 Example: E_{TM} is undecidable

In this example, we will show that a turing machine that accepts the empty language is undecidable.

$$E_{TM} = \{\langle M \rangle \mid \mathcal{L}_M = \emptyset\}$$

This is a proof by contradiction. First, suppose D decides E_{TM}: We are looking for a transformation, $\langle M, \omega \rangle \Rightarrow X_{M,\omega}$ such that $\mathcal{L}_X = \emptyset \iff M$ does not accept ω.

Now let's designate $X_{M,\omega}$. On input ω': Simulate M on input ω. If M accepts, accept. Otherwise, reject.

This description leads us to the following observations:

- If $\langle M, \omega \rangle \in A_{TM}$ then $\mathcal{L}(X_{M,\omega}) = \Sigma^*$ and,
- If $\langle M, \omega \rangle \notin A_{TM}$ then $\mathcal{L}(X_{M,\omega}) = \emptyset$

The next step may require a little leap of faith (or alternatively, a lot of insight). Here we go: given M and ω we can create a description of X and, as it turns out, the algorithm below shows us how to create a decider, R, for A_{TM}.

$R =$ On input $\langle M, \omega \rangle$: create $\langle X_{M,\omega} \rangle$ and then simulate D on input $\langle X_{M,\omega} \rangle$. If D accepts, reject. Otherwise, accept.

We have now simulated R, a decider for A_{TM}. If D decides E_{TM} then R decides A_{TM}. But we proved a A_{TM} is not decidable, so this is a contradiction. Therefore, we know E_{TM} is undecidable by contradiction.

Note: We can alter the method above slightly to construct a proof that shows E_{TM} is unrecognizable.

Change the line:

> *This is a proof by contradiction. First, suppose D* **decides** *E_{TM}:*
>
> to
>
> *This is a proof by contradiction. First, suppose D* **recognizes** *E_{TM}:*

Then, in the second step of our construction of a decider D for A_{TM}, change

> *Simulate D on input $\langle X_{M,\omega} \rangle$. If D accepts,* **reject**. *Otherwise,* **accept**.
>
> to
>
> *Simulate D on input $\langle X_{M,\omega} \rangle$. If D accepts,* **accept**. *Otherwise,* **reject**.

It follows that if $\omega \in \mathcal{L}_M$, we reject. If $\omega \notin \mathcal{L}_M$, we accept. But we proved $\overline{A_{TM}}$ is not recognizable, so by contradiction E_{TM} is not recognizable.

2.7.2 Example: EQ_{TM} is undecidable

$$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid L_{M_1} = L_{M_2}\}$$

Suppose that Z is a TM such that $L_Z = \emptyset$.

Then the question of whether $\langle M, Z \rangle \in EQ_{TM}$ is equivalent to the question of whether $\langle M \rangle \in E_{TM}$.

Suppose D decides EQ_{TM}. We can describe D in the following manner:

$D =$ On input $\langle M \rangle$, produce $\langle M, Z \rangle$ and then run D on $\langle M, Z \rangle$. If D accepts, accept. Otherwise, reject.

If $\langle M \rangle \in E_{TM}$ we accept. If $\langle M \rangle \notin E_{TM}$ we reject. But we know that E_{TM} is undecidable, thus EQ_{TM} is undecidable.

Alternatively, we can use A_{TM} instead of E_{TM} to prove the same thing by changing our construction of Decider D to be as follows:

$D =$ On input $\langle M, w \rangle$, produce $\langle X_{M,w} \rangle$, produce $\langle X_{M,w}, Z \rangle$ and then run D on $\langle X_{M,w}, Z \rangle$. If D accepts, reject. Otherwise, accept.

If D decides EQ_{TM} it must also decide A_{TM}. But we know A_{TM} is not decidable, thus EQ_{TM} must not be decidable.

2.7.3 Mapping reducibility

Mapping reducibility allows us to formalize reducibility. When you can reduce a problem A to another problem B, then a function exists that can convert instances of problem A to problem B. This function is called a *reduction.* We can used the reduction to solve A with a solver for B, because any instance of A can be solved by using the reduction to convert it to an instance of B.

A TM computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape. (*cf.* section 2.1.1). We say that a computable function is a function on strings such that there is a TM that on input x halts with $f(x)$ on the tape.

Definition Language A is *mapping reducible* to language B, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \longrightarrow \Sigma^*$, where for every w, $w \in A \Longleftrightarrow f(w) \in B$. The function f is called the *reduction* of A to B.

In other words, a mapping reduction is a relationship between languages. We say that A *mapping reduces* to B (or $A \leq_m B$) if there is a computable function f such that $\forall x \in A$, $f(x) \in B$ and $\forall x \notin A$, $f(x) \notin B$ (*i.e.*, $\forall x, x \in A \Longleftrightarrow f(x) \in B$).

The following are some interpretations of the meaning of $A \leq_m B$: A is easier to figure out than B. If we can solve B then we can solve A. In other words, A is easier than B because if we can solve B we can solve A by applying f then determining if the output is in B. B is more useful or more general than A.

Examples

- $E_{TM} = \{\langle M \rangle \mid \mathcal{L}(M) = \phi\}$
  - $\overline{A_{TM}} \leq_m E_{TM}$ - if you can decide/recognize E_{TM}, you can with $\overline{A_{TM}}$ as well
  - To show this, you must find $f : f(\langle M, w \rangle) \in E_{TM} \Longleftrightarrow \langle M, w \rangle \in \overline{A_{TM}}$. This means that you need to find the function that will mapping reduce items in $\overline{A_{TM}}$ to E_{TM}.
  - $f : \langle M, w \rangle \longrightarrow \langle X_{M,w} \rangle$ where $X_{M,w}$ is such that on input w': Run M on input w, accept if it accepts, reject otherwise.
  - f is computable
  - $\langle M, w \rangle \in \overline{A_{TM}} \Rightarrow f(\langle M, w \rangle) \in E_{TM}$ since in this case $\mathcal{L}(X_{M,w}) = \emptyset$
  - $\langle M, w \rangle \in \overline{A_{TM}} \Leftarrow f(\langle M, w \rangle) \in E_{TM}$, since if M accepts w, $\mathcal{L}(X_{M,w}) = \Sigma^* \neq \emptyset$.
- $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid \mathcal{L}(M_1) = \mathcal{L}(M_2)\}$
  - Let Z be a Turing machine that always rejects (so, $\mathcal{L}(Z) = \phi$)
  - Let $X_{M,w}$ be: On input w', run M on w, if M accepts w, accept, otherwise reject.
  - Suppose R recognizes EQ_{TM} then R' on input $\langle M, w \rangle$ runs as follows:

1. Produce the pair $\langle X_{M,w}, Z\rangle$
2. Run R on this; if R accepts, accept; otherwise, reject.

  – Claim R' recognizes $\overline{A_{TM}}$.
  – If $\langle M, w\rangle \in \overline{A_{TM}}$, accept.
  – If $\langle M, w\rangle \notin \overline{A_{TM}}$ (in other words, $\langle M, w\rangle \in A_{TM}$), do not accept.
  – Therefore there is no R. Therefore EQ_{TM} is not recognizable.
  – So $f : \langle M, w\rangle \longrightarrow \langle X_{M,w}, Z\rangle$
  – If $\langle M, w\rangle \in \overline{A_{TM}}$, $\mathcal{L}(X_{M,w}) = \phi = \mathcal{L}(Z)$ so $f(\langle M, w\rangle) \in EQ_{TM}$
  – If $\langle M, w\rangle \notin \overline{A_{TM}}$, $\mathcal{L}(X_{M,w}) = \Sigma^* \neq \phi = \mathcal{L}(Z)$ so $f(\langle M, w\rangle) \notin EQ_{TM}$

- $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$; $\overline{EQ_{TM}} = \{\langle M_1, M_2\rangle \mid \mathcal{L}(M_1) \neq \mathcal{L}(M_2)\}$
  – A is a TM that always accepts
  – *mapping reduction:* $f : \langle M, w\rangle \longrightarrow \langle X_{M,w}, A\rangle$
  – Suppose R recognizes $\overline{EQ_{TM}}$
  – Then R': on input $\langle M, w\rangle$
    1. Produce $\langle X_{M,w}, A\rangle$
    2. Run R on this. If it accepts, accept; otherwise, reject.
  – Claim: R' recognizes $\overline{A_{TM}}$
    * If $\langle M, w\rangle \in \overline{A_{TM}}$, R accepts, so we accept
    * If $\langle M, w\rangle \in A_{TM}$, R does not accept, so neither do we
  – Therefore, EQ_{TM} is not recognizable.

- Let $L = \{\langle M\rangle \mid M \text{ accepts } 0\}$. Prove $A_{TM} \leq_m L$.
  – $f : \langle M, w\rangle \longrightarrow \langle X_{M,w}\rangle$, such that M accepts w if and only if $X_{M,w}$ accepts 0. You want it to work out so that if M accepts w, then $X_{M,w}$ accepts 0, and if M does not accept w, then $X_{M,w}$ does not accept 0.
  – $X_{M,w}$: on input w':
    1. if $w' \neq 0$, reject
    2. otherwise (if $w' = 0$) run M on input w, if it accepts, accept; otherwise, reject.
  – Assume L is decidable
  – D decides L
  – D' on input w'
    1. Produce $\langle X_{M,w}\rangle$
    2. Run D on this. If it accepts, accept; otherwise, reject.
  – This is a decider for A_{TM}, but there can be no such decider.

- $L = \{\langle M_1, M_2\rangle \mid \mathcal{L}(M_1) \not\subseteq \mathcal{L}(M_2) \text{ and } \mathcal{L}(M_2) \not\subseteq \mathcal{L}(M_1)\}$

- This means that neither language can be empty or Σ^*. There must be some elements in L_1 that are not in L_2 and there must be some elements in L_2 that are not in L_1.
- Prove L is undecidable.
- Prove $A_{TM} \leq_m L$ $(\langle X_{M,w}, M_2 \rangle)$
- $f : \langle M, w \rangle \longrightarrow \langle M_1, M_2 \rangle$
- M_2 is a machine accepting 0^*
- $X_{M,w}$:
  1. We will accept 1^* if M accepts w
  2. We will accept only ϵ if M does not accept w.

  Thus, $X_{M,w}$'s language is a subset of M_2's if M does not accept w, but if M accepts w then neither is a subset of the other.

Summary A *mapping reduction* of A to B ($A \leq_m B$) is a way to convert questions about membership testing in A to membership testing in B, which is an easier, more general problem. To find out if $w \in A$, we can use f to map w to $f(w)$ and see if $f(w) \in B$. A *mapping reduction* describes the mapping that creates the reduction.

Chapter 3

Recursive functions

3.1 Primitive recursive functions

What is the expressiveness power of a given programming language? To answer this question, in the quest for a model of what we understand by calculus, we will build a set of functions that we call *Primitive Recursive Functions* (PRF).

We start with a collection of functions of which its simplicity make it impossible to doubt about its computability. This set of functions, which will be used as basis for construct others, will be called Basis Functions. Then we will show that we can combine those basis functions to form others of which its computability is derived from the originals.

3.1.1 Numeric functions

Definition A *numeric function* is a function such that its domain is a Cartesian power of $\mathbb{N}_0$ and its target is $\mathbb{N}_0$, *i.e.* $f : \mathbb{N}_0^k \to \mathbb{N}_0$ with $k \in \mathbb{N}_0$.

Notations We denote by $f^{(k)}$ the function $f : \mathbb{N}_0^k \to \mathbb{N}_0$. Also, we denote by X, Y, Z the kth-tuples of $\mathbb{N}_0^k$, and when we what to emphasize that there are k elements we denote it by X^k. From now on, unless we specify the contrary, all the numbers are considered elements of $\mathbb{N}_0$. As a last remark, we follow the convention that a function of zero variables, $f : \mathbb{N}_0^0 \to \mathbb{N}_0$ denotes one element of its target, so we can refer to the elements of $\mathbb{N}_0$ by referring to functions of type $\mathbb{N}_0^0 \to \mathbb{N}_0$.

3.1.2 Basis functions

The base of the hierarchy of computable functions is a family of functions, which will be the key to defining the PRF.

Successor The *successor* function is defined by

$$s : \mathbb{N}_0 \to \mathbb{N}_0$$

$$x \in \mathbb{N}_0 \mapsto s(x) \overset{def}{=} x + 1 \in \mathbb{N}_0$$

That is, s produces the successor to its input value. Looking at it this way, s is a computable function, since we already know a computable process for the sum of integers.

Zero The *zero* function is defined by

$$z : \mathbb{N}_0 \to \mathbb{N}_0$$

$$x \in \mathbb{N}_0 \mapsto z(x) \overset{def}{=} 0 \in \mathbb{N}_0$$

We can easily accept that z is computable, since it only replaces a value by 0.

Projections The *projection of the k-th component*, or *projection* function is defined by

$$u_k^{(n)} : \mathbb{N}_0^n \to \mathbb{N}_0$$

$$(x_1, x_2, \ldots, x_n) \mapsto u_k^{(n)}(x_1, x_2, \ldots, x_n) \overset{def}{=} x_k \in \mathbb{N}_0$$

Where $n, k \in \mathbb{N}$ with $1 \geq k \geq n$.

For example $u_3^{(4)}(x, y+1, f(x,y), f(x,y+1)) = f(x,y)$

Analogously to previous cases, it is easy to proof that projections are in the class of computable functions.

We refer to $s^{(1)}$, $z^{(1)}$ and all the projections $u_k^{(n)}$ with $1 \leq k \leq n$ as *basis functions*.

The functions presented at this section cannot do so much by themselves, so we will study how them can be used to make more complex functions.

3.1.3 The composition

Given a numeric function $f^{(n)}$ and a family of numeric functions $\{g_i^{(m)}\}_{i=1}^n$, we define a new function $h^{(m)}$ as

$$h : \mathbb{N}_0^m \to \mathbb{N}_0$$

$$X \in \mathbb{N}_0^m \mapsto h(X) \overset{def}{=} f(g_1(X), g_2(X), \ldots, g_n(X)) \in \mathbb{N}_0$$

Is easy to see that the composition of computable functions is computable, since being f computable and also the family g_i ($i \in \mathbb{N}$), it suffices to calculate all the g_i and use its outputs as input for f.

Notation $h = \Phi(f, g_1, g_2, \ldots, g_n)$

Example Let us define the function $one : \mathbb{N}_0 \to \mathbb{N}_0$, such that $\forall x,\ one(x) = 1$. It is easy to see that $one = \Phi(s, z)$. In the same way $two = \Phi(s, \Phi(s, z)) = \Phi(s, one)$.

Continuing with the same process, all constant function can be expressed with the basis functions and the composition.

3.1.4 The recursion

Finally, we will see the a constructor called *recursion*. This, combined with the previous ones, will allows as to represent many computable functions.

Definition Let $k \in \mathbb{N}_0$ and let $g^{(k)}$ and $h^{(k+2)}$ be two numeric functions. We define a new function $f^{(k+1)}$ in the following way

$$f(X^k, 0) \overset{def}{=} g(X^k)$$

$$f(X^k, y+1) \overset{def}{=} h(X^k, y, f(X^k, y))$$

Notation $f = R(g, h)$.

We can see that f is well defined. Notice that the definition allows the possibility of k being equal to zero. In this case we find that g have to accept zero variables, which by convention we identify it with elements of N.

To conclude, notice that a function constructed by recursion over computable functions must be considered computable. If f is defined by recursion over two computable functions g and h, we can calculate $f(X, y)$ by calculating first $f(X, 0) = g(X)$, which is computable, then $f(X, 1)$, then $f(X, 2)$ and so on until $f(X, y)$ is reached.

Example Let us define the function $\Sigma(x)$ (sum) such that $\Sigma(x, y) \overset{def}{=} x + y$. We have

$$\Sigma(x, 0) = x + 0 = x = u_1^{(1)}(x)$$

Also,

$$\begin{aligned}\Sigma(x, y+1) = x + (y+1) = (x+y) + 1 = s(\Sigma(x, y)) \\ = s(u_3^{(3)}(x, y, \Sigma(x, y))) = \Phi(s, u_3^{(3)})(x, y, \Sigma(x, y))\end{aligned}$$

So we say we can construct Σ by recursion over $u_1^{(1)}$ and $\Phi(s, u_3^{(3)})$, moreover,

$$\Sigma = R(u_1^{(1)}, \Phi(s, u_3^{(3)}))$$

Now we can define the set of primitive recursive functions. We do it by induction.

Primitive recursive functions

1. The basis functions defined in section 3.1.2 are *primitive recursive functions.*
2. Functions obtained from the basis functions by applying a finite number of compositions and recursions are *primitive recursive functions.*
3. Those are all the *primitive recursive functions.*

We denote by **PRF** to the set of primitive recursive functions.

Power of a function Given a function $f^{(1)}$, we define a new function $F^{(2)}$, called *power of* f, such that $\forall x \in \mathbb{N}_0,\ F(x,0) = x$ and if $y \in \mathbb{N}_0,\ F(x,y+1) = f(F(x,y))$. Notation $f^y(x) \overset{def}{=} F(x,y)$.

Theorem 11 *If* $f \in \mathbf{PRF}$*, then* $F \in \mathbf{PRF}$.

Proof $F = R(u_1^{(1)}, \Phi(f, u_3^{(3)}))$.

3.1.5 Examples of primitive recursive functions

The following function belong to the set **PRF**.

1. $Pd(x) \overset{def}{=} \begin{cases} 0 & \text{if } x = 0 \\ x-1 & \text{in other case} \end{cases}$
2. $\Pi(x,y) \overset{def}{=} xy$
3. $Fac(x) \overset{def}{=} x!$
4. $Exp(x,y) \overset{def}{=} x^y$ with the convention that $0^0 = 1$
5. $E(x,y) \overset{def}{=} \begin{cases} 1 & \text{if } x = y \\ 0 & \text{in other case} \end{cases}$

We show only the first example. The rest is left as **exercise**.

We want to construct a function $Pd(x)$ which follows the specification 1. Notice that $Pd(0) = 0 = z^{(0)}$, also $Pd(y+1) = y = u_1^{(2)}(y, f(y))$. Then we can deduce that we can construct Pd by recursion over $z^{(0)}$ and $u_1^{(2)}$ as $Pd = R(z^{(0)}, u_1^{(2)})$. Then $Pd \in \mathbf{PRF}$.

3.1.6 Primitive recursive sets (PRS)

Characteristic function Given a set X, for every subset $A \subseteq X$ we define its characteristic function $\chi : X \to \{0,1\}$ by

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

Notice that the characteristic function of a set fully determines it. Inversely, given a function $\chi : X \to \{0,1\}$, there exists a unique subset $A \subseteq X$ such that $\chi_A = \chi$. The following theorem formalizes this idea

Theorem 12 *Let* X *be a set,* $\mathcal{P}(X)$ *its power set and* $\mathcal{F}$ *the set of all the functions* $\chi : X \to \{0,1\}$*. Then the application*

$$\Psi : \mathcal{P}(X) \to \mathcal{F}$$

$$A \in \mathcal{P}(X) \mapsto \Psi(A) \overset{def}{=} \chi_A \in \mathcal{F}$$

is a bijection, of which its inverse is given by

$$\Phi : \mathcal{F} \to \mathcal{P}(X)$$

$$\chi \in \mathcal{F} \mapsto \Phi(\chi) \overset{def}{=} \{x \in X : \chi(x) = 1\} \in \mathcal{P}(X)$$

Proof It suffices by showing that $\Psi\Phi = Id_{\mathcal{F}}$ and $\Phi\Psi = Id_{\mathcal{P}(X)}$ which is trivial.

What this theorem tells us is that it is essentially the same work with subsets of X or with functions in $\mathcal{F}$.

Our interest in characteristic functions is that they allow us to express boolean operations between sets in an algebraic way. Moreover, we can stand the following theorem.

Theorem 13 *Let X be a set. For each $A, B \subseteq X$, one has*

1. $\chi_{A\cup B} = \chi_A + \chi_B - \chi_A{\cdot}\chi_B$
2. $\chi_{A\cap B} = \chi_A{\cdot}\chi_B$
3. $\chi_{X\setminus A} = 1 - \chi_A$

Proof Exercise

This idea is linked to the **PRF** by the following definition

Primitive recursive sets Let $k \in \mathbb{N}$. We say that a subset $A \subseteq \mathbb{N}_0^k$ is a *primitive recursive set* if its characteristic function $\chi_A : \mathbb{N}_0^k \to \{0,1\}$ is a primitive recursive function.

With this definition plus theorem 13, we can stand the following theorem.

Theorem 14 *Let $k \in \mathbb{N}$ and let $A, B \subseteq \mathbb{N}_0^k$. If A and B are primitive recursive sets, then $A \cup B$, $A \cap B$ and $\mathbb{N}_0^k \setminus A$ are primitive recursive sets.*

Proof Exercise

Further results

Theorem 15 *Let $k \in \mathbb{N}$. All the finite subsets of $\mathbb{N}_0^k$ are primitive recursive.*

Proof Let $A \subset \mathbb{N}_0^k$ a finite subset. Since $A = \bigcup_{x\in A} \{x\}$, it suffices to show that the empty set and all the one-element subsets of $\mathbb{N}_0^k$ are primitive recursive.

Let's start with the empty set. We know that $\forall X \in \mathbb{N}_0^k,\ \chi_\emptyset(X) = 0$. So $\chi_\emptyset = z^{(k)} \in$ **PRF**.

Now will see what happen with sets of exactly one element. We proceed by induction on k. Let $k = 1$ and $a \in \mathbb{N}_0$. We must show that $A = \{a\} \subset \mathbb{N}_0$ is a primitive recursive set. To do so, we only have to verify that $\chi_A = \Phi(E, u_1^{(1)}, \mathrm{a}^{(1)})$ where $\mathrm{a}^{(1)}$ is the constant function of one variable of value a and E is defined in section 3.1.5, which is trivial.

Now assume $k \in \mathbb{N}$ such that all the one-element sets of $\mathbb{N}_0^k$ are primitive recursive. We have to show the case $k+1$. Let $a = (a_1, a_2, \ldots, a_{k+1}) \in \mathbb{N}_0^{k+1}$ and let $b = (a_1, a_2, \ldots, a_k)$. We know that $\chi_{\{b\}}^{(k)}$ is **PRF**. Then

$$\chi_{\{a\}}^{(k+1)} = \Phi(\Pi, \Phi(\chi_{\{b\}}, u_1^{(k+1)}, \ldots, u_k^{(k+1)}), \Phi(E, u_{k+1}^{(k+1)}, a_{k+1}))$$

which tell us that $\chi_{\{a\}}$ is **PRF**.

Theorem 16

1. *Let $m \in \mathbb{N}$ and let $A, B \subset \mathbb{N}_0^m$ be primitive recursive sets such that $A \cup B = \mathbb{N}_0^m$ and $A \cap B = \emptyset$,* i.e. *A and B form a partition of $\mathbb{N}_0^m$. Let also $f^{(m)}$ and $g^{(m)}$ be primitive recursive functions. Then the function*

$$h^{(m)}(X) \overset{def}{=} \begin{cases} f(X) & \text{if } X \in A \\ g(X) & \text{if } X \in B \end{cases}$$

   is primitive recursive.

2. *If $\{A_i\}_{i=1}^{l} \subset \mathcal{P}(\mathbb{N}_0^m)$ is a finite partition of $\mathbb{N}_0^k$, where $\forall i,\ A_i \in \mathbf{PRS}$, and also $\{f_i^{(m)}\}_{i=1}^{l} \subset \mathbf{PRF}$ is a family of primitive recursive functions, then the function*

$$F^{(m)} \overset{def}{=} f_i(X) \text{ if } X \in A_i$$

   is a primitive recursive function.

Proof

1. Let $h = f.\chi_A + g.\chi_B = \Phi(\Sigma, \Phi(\Pi, f, \chi_A), \Phi(\Pi, g, \chi_B))$. Since A and B are **PRS**, χ_A and χ_B are **PRF**, so h is **PRF**.

2. **Exercise**.

Corollary 17 *Let $f^{(m)}$ be a numeric function such that $\{X \in \mathbb{N}_0^m : F(X) = 0\}$ is finite. Then $f \in \mathbf{PRF}$.*

Proof It follows from previous theorem and the fact that one-element sets are primitive recursive.

3.1.7 Primitive recursive relations (PRR)

Finally, we introduce the primitive recursive relations. Remind that a relation R between two sets A and B is such that $R \subseteq A \times B$.

Primitive recursive relations A relation $R \subseteq \mathbb{N}_0 \times \mathbb{N}_0$ is *primitive recursive* if the defined set if primitive recursive.

For the previous definition we can deduce that a relation is primitive recursive if the characteristic function of the defined set is primitive recursive.

Example Let's show that the relation '=' is primitive recursive. The equal relation defines the set $R_= = \{(x, x), x \in \mathbb{N}_0\}$. The idea is to find a primitive recursive function f which characterize the set $R_=$. Notice that the function $E(x, y)$ defined in section 3.1.5 is the one that we are looking for. So $f_{R_=} = E$ and then the relation '=' is primitive recursive.

3.2 Beyond PRF

In our quest to find a model of calculus we have defined the set of primitive recursive functions, and after certain amount of examples we could think that **PRF** is the model we were looking for. Would it be any function which we cannot represent as primitive recursive? We will see later that there are. To better understand why, we will start looking at the characteristics of the primitive recursive functions.

3.2.1 Characteristics of PRF

Theorem 18 *If f is* **PRF***, then f is a total function.*

Proof Notice that basis functions are defined for all the values of $\mathbb{N}_0$, so basis functions are total functions. In addition, the composition and recursion processes are conservative with the totalness property (**exercise**), so functions in **PRF** are total functions.

Notice that there are functions which are no total in $\mathbb{N}_0$. As instance, the numeric function which calculates the square root of a number. Then we find that there are computable functions which cannot be represented by a **PRF**.

Furthermore, in 1928 Wilhelm Ackermann showed a total, computable function which is not primitive recursive. Nowadays that functions is known as Ackermann function.

3.2.2 The Ackermann sequence

To show that there are total functions which are not **PRF**, we will look for a property which all the **PRF** have, and then we will look for a total function which has not such a property.

First, let us define the following sequence of functions, which will be called Ackermann functions:

$$f_0(x) = s(x)$$

$$f_1(x) = f_0^{x+2}(x) = s^{x+2}(x) = 2x + 2$$

$$f_2(x) = f_1^{x+2}(x) = (s^{x+2})^{x+2}(x) = 2(\ldots 2(2(2x+2)+2)+2\ldots)+2$$

$$\vdots$$

$$f_{k+1}(x) = f_k^{x+2}(x)$$

where f^n is the power function (the function which applies f, n times).

Theorem 19 (Properties)

1. *$\forall k,\ f_k \in$ **PRF***
2. *$x > x' \Rightarrow f_k(x) > f_k(x')$*
3. *$\forall x, k \Rightarrow f_k(x) > x$*
4. *$\forall x, k \Rightarrow f_{k+1}(x) > f_k(x)$*

Proof

1. Induction over k. Base case, $k = 0$, notice that $f_0(x) = s(x) \in \mathbf{PRF}$. Assume $f_k \in \mathbf{PRF}$, then notice that $f_{k+1}(x) = f_k^{x+2}(x)$, which by property of power, is a **PRF**.

2. **Exercise** (Tip: Induction over k)

3. Induction over k. Base case, $k = 0$, $f_0(x) = s(x) > x$. Assume is true for k, so $f_k(x) > x$, we must prove $f_{k+1}(x) > x$.

$$f_{k+1}(x) = f_k^{x+2}(x) = f_k(f_k^{x+1}(x)) \overset{(IH)}{>} f_k^{x+1}(x) = f_k(f_k^{x}(x)) \overset{(IH)}{>} f_k^{x}(x) =$$
$$\vdots$$
$$= f_k^2(x) = f_k(f_k(x)) \overset{(IH)}{>} f_k(x) \overset{(IH)}{>} x$$

4. $f_{k+1}(x) \overset{(a)}{=} f_k^{x+2}(x) \overset{(b)}{=} f_k^{x+1}(f_k(x)) \overset{(c)}{>} f_k^{x+1}(x) > \ldots > f_k(x)$
   (a) By definition of the Ackermann function.
   (b) By definition of the power function.
   (c) Direct from items 2 and 3

Majoration We say that a function $f^{(1)}$ *majorates* a function $g^{(n)}$ if

$$\forall x_1, x_2, \ldots, x_n,\ f(\max(x_1, x_2, \ldots, x_n)) \geq g(x_1, x_2, \ldots, x_n)$$

Notation: $f^{(1)} \to g^{(n)}$

Theorem 20 *Let $g^{(n)} \in \mathbf{PRF}$. Then there exists a f_k in the Ackermann sequence such that $f_k \to g^{(n)}$.*

Proof Taking advantage of the inductive definition of **PRF** we show this property by induction. First we prove basis functions have this property (1), then we prove that it is preserved by composition (2) and by recursion (3).

1. The proof that basis functions are majorated by f_0 is left as **exercise**.

2. We want to prove that if $g^{(n)} = \Phi(I^{(m)}, h_1^{(n)}, h_2^{(n)}, \ldots, h_m^{(n)})$, then

$$f_k \to I^{(m)} \ \wedge\ \forall i, f_k \to h_i^{(n)} \Rightarrow f_{k+1} \to g^{(n)}$$

First, notice that if $h_1^{(n)}(X) \leq f_k(\max(X)), \ldots h_m^{(n)}(X) \leq f_k(\max(X))$ then

$$\max\{h_1^{(n)}(X), h_2^{(n)}(X), \ldots, h_m^{(n)}(X)\} \leq f_k(\max(X)) \qquad (3.1)$$

Then

$$g(X) \overset{def}{=} I(h_1(X), \ldots, h_n(X)) \overset{3.1}{\leq} f_k(\max(h_1(X), \ldots, h_n(X))) \leq$$
$$\overset{(a)}{\leq} f_k(f_k(\max(X))) \overset{(b)}{\leq} f_k^{\max(X)+2}(\max(X)) \overset{def}{=} f_{k+1}(\max(X))$$

(a) By 3.1 and property 2.

(b) By successive applications of properties 2 and 3.

3. We want to prove that if $g^{(n+1)} = R(I^{(n)}, h^{(n+2)})$, then

$$f_k \to I \ \wedge \ f_k \to h \Rightarrow f_k \to g^{(n)}$$

First, we know by definition of R that $g(X,0) = I(X)$ and $g(X, y+1) = h(X, y, g(X,y))$ and by hypothesis we have

$$g(X,0) = I(X) \leq f_k(\max(X)) \tag{3.2}$$

Also

$$g(X,1) = h(X,0,g(X,0)) \overset{hyp}{\leq} f_k(\max(X,0,g(X,0))) \leq$$

$$\overset{(a)}{\leq} f_k(\max(f_k(\max(X)),0,X)) \overset{(b)}{\leq} f_k(f_k(\max(X)))$$

(a) By property 2 and 3.2.

(b) $\forall k, f_k(\max(X)) > \max(X,0)$.

It can be proven by induction that $\forall y, g(X,y) \leq f_k^{(y+1)}(\max(X))$ and notice that

$$f_k^{y+1}(\max(X)) \overset{(a)}{\leq} f_k^{y+1}(\max(y,X)) \overset{(b)}{\leq} f_k^{\max(y,X)+1}(\max(y,X)) \leq$$

$$f_k^{\max(y,X)+2}(\max(y,X)) = f_{k+1}(\max(y,X))$$

(a) Adding one element to the set, it can be equal in case y is not the max or bigger in other case.

(b) By property 2.

So, we conclude that $g(X,y) \leq f_{k+1}(\max(y,X))$.

From 1, 2 and 3 we can conclude that the theorem holds, since every $f \in$ **PRF** is obtained by applying a finite number of operators R and Φ to the basis functions.

Despite the fact that all the functions in the Ackermann sequence are **PRF** (*cf.* property 1), we can use them to define a function which is not in **PRF**.

The Ackermann function We define the *Ackermann* function by

$$ACK(x) = f_x(x)$$

Notice that this function is computable, since each f_x is, so we just have to find the x-th one and compute it with argument x. However it is not **PRF**, as stated in the following theorem.

Theorem 21 $ACK(x) \notin$ **PRF**

Proof Assume $ACK(x) \in \mathbf{PRF}$, then $ACK(x)+1 \in \mathbf{PRF}$, and so by theorem 20, there exists k such that $\forall x$, $ACK(x)+1 \leq f_k(x)$. In particular, it must be also valid for $x = k$, so $ACK(k)+1 \leq f_k(k) = ACK(k)$, which is a contradiction.

So, $ACK(x)$ is not majored by any function in the Ackermann sequence, a fundamental property of **PRF**, so $ACK(x)$ is a computable, total function, which is not **PRF**.

3.3 Recursive functions

In the previous section we just proved that there exists total functions which are not primitive recursive. In this section we add a new operator in order to define a new family of functions, which will be called recursive functions.

3.3.1 The minimizer

We add to the previous constructors (namely, the composition and the recursion), a new operator, called *the minimizer.*

The minimizer Given $f^{(n+1)}$, we say that $g^{(n)}$ is constructed by minimization of f, notation $M[f]$ if

$$g^{(n)}(X) = M[f](X) = \mu_t(f(t,X) = 0)$$

That is, $g(X)$ is the minimum t such that $f(t,X) = 0$.

Example We define the partial function *root* such that $root(x) = \sqrt{x}$. Notice that this function is not defined for values of x such that $\sqrt{x} \notin \mathbb{N}_0$.

$$root(x) = \sqrt{(}x) = \min\{t \in \mathbb{N}_0 \mid t^2 = x\}$$

Notice that

$$t^2 = x \iff E(t^2, x) = 1 \iff {}^{\circ}D(E(t^2, x)) = 0$$

where ${}^{\circ}D(x) \overset{def}{=} 1$ if $x = 0$ or 0 in other case (*cf.* exercise sheet).
Then $g(x) = \mu_t({}^{\circ}D(E(t^2, x)) = 0) = M[\Phi[{}^{\circ}D, \Phi[E, \Phi[\Pi, u_1^{(2)}, u_1^{(2)}], u_2^{(2)}]]](x)$

Now we can define the recursive functions.

Recursive functions We define inductively the set **RF** of recursive functions as follows

1. If $f \in \mathbf{PRF}$ then $f \in \mathbf{RF}$
2. If $f^{(n)} \in \mathbf{RF}$, then $g^{(n-1)} = M[f] \in \mathbf{RF}$.
3. No other functions are in **RF**.

3.4 The Church thesis

Church Thesis *A function of positive integers is effectively calculable only if recursive*

This thesis, formulated by Alonso Church, is equivalent to the one presented in section 2.2. What we are saying is, every computable function (in terms of Turing Machines as defined in the previous chapter) is in **RF**.

3.5 Beyond RF

Are there natural functions which are not **RF**? According to the Turing-Church thesis it is equivalent to ask if there are non-calculable natural functions. The answer is YES.

Notice that, by definition, every **RF** is identified by a unique formula. To write this formula, we use a numerable set of symbols. The finite combinations of subsets of a numerable set is a numerable set. Notice that not every combination of symbols is a **RF**, so we can conclude that the cardinality of combinations of symbols ($\aleph_0$) is the same or bigger than the cardinality of **RF**. So the cardinality of **RF** formulas is at most $\aleph_0$.

Now we can calculate the cardinality of natural functions. It is easy to see that the functions with $\{0, 1\}$ as image set, form a subset of the set of natural functions. These functions are known as characteristic functions, which have a biunivocal correspondence with the set $\mathcal{P}(\mathbb{N})$. The cardinality of $\mathcal{P}(\mathbb{N})$ is $\aleph_1$, therefore, the cardinality of the set of characteristic functions is $\aleph_1$. Since the set of characteristic functions is a subset of the natural functions, we can conclude that the last set has also cardinality $\aleph_1$.

Then, there exists natural functions which are not recursive functions.

Chapter 4

Complexity: computable functions in practice

During World War II, Alan Turing helped design and build a specialized computing device called the Bombe at Bletchley Park. He used the Bombe to crack the German "Enigma" code, greatly aiding the Allied cause. By the 1960's computers were widely available in industry and at universities. As algorithms were developed to solve myriad problems, some mathematicians and scientists began to classify algorithms according to their efficiency and to search for best algorithms for certain problems. This was the beginning of the modern theory of computation.

In this section we are dealing with *complexity* instead of *computability*, and all the Turing machines that we consider will halt on all their inputs.

The time that an algorithm takes depends on the input and the machine on which it is run. The first important insight in complexity theory is that a good measure of the complexity of an algorithm is its asymptotic worst-case complexity as a function of the size of the input, n.

Worst-case complexity For an input, ω, let $n = |\omega|$ be the length of ω. We say that a Turing machine M runs in time $T(n)$ if for all ω of length n, $M(\omega)$ takes at most $T(n)$ steps and then halts. This is called *worst-case complexity* because $T(n)$ must be as large as the time taken by any input of length n.

Big O notation Upper and lower bounds are usually stated using the big O notation, which hides constant factors and smaller terms. This makes the bounds independent of the specific details of the computational model used. For instance, if $T(n) = 7n^2 + 15n + 40$, in big O notation one would write $T(n) = \mathcal{O}(n^2)$.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

Example 12 *Let M be a Turing machine that accepts the regular language $L = \{a^n b^n \mid n \geq 1\}$.*

This Turing machine starts and stays in state p when reading a's. It changes a to x during a left-to-right scan. When it encounters the first b, it changes the

b to y and moves from right to left to find the last x. This last x is changed to y and the machine again moves from left to right to locate the next b.

After all b's are examined, it changes to state s and moves from right to left to the blank just before the input. At this time, the machine enters state t, which is the only final state.

See example 5 for a low-level description.

What is its complexity? *First will see what happen with an accepted word as input, say $a^n b^n$, so the size of the input is $N = 2n$. The first step changes each a by x until it encounters a b, so it takes n steps to do so. At the b it changes it by y (1 step) and moves right to find the last x (1 step). Then it moves to right to find the next b, so 2 steps, and do the same: moves to left which takes 3 steps to find the first x, then to right again 4 steps, to left 5 steps, to right 6 and so on. The last move to the right takes $2n - 1$ steps. At this point the machine have to move to the first position again, which takes $2n$ more steps. So we have $n + 1 + (1 + 2 + 3 + \ldots + 2n - 1) + (2n)$ steps, which is the same to $n + 1 + \sum_{i=1}^{2n} i = n + 1 + \frac{2n(2n+1)}{2} = 2n^2 + 2n + 1 = \frac{1}{2}N^2 + N + 1$. So we use the big O notation to say that the complexity of this machine with an accepted word is given by $\mathcal{O}(N^2)$.*

Now will see what happen with a non accepted word. Say $a^m b^n$ with $m > n$. The size of this word is $N = n + m < 2m = N'$. This machine will do all the process until it match the last b with an a, at this point it halts (in a non-final state). Then the number of steps is given by $m + 1 + (1 + 2 + \ldots + 2n - 1) = 2n^2 - n + m + 1 < 2m^2 + 1 = \frac{1}{2}N'^2 + 1$. So we take $\mathcal{O}(N'^2)$ as an upper bound of the complexity in this case.

Similarly, if the input word is $a^n b^m$ with $m > n$, we get $2m^2 - m + n + 1 < 2m^2 + 1$ steps, so again the upper bound is given by $\mathcal{O}(N'^2)$.

So, in the general case, if the input is a succession of a's and b's, the upper bound complexity is given by $\mathcal{O}(N^2)$ being N twice the maximum between the amount of a's and b's, i.e., *being N an upper bound of the input size.*

Notice that other input configurations can be detected faster, and so, the upper bound complexity holds.

Complexity classes Let $f : \mathbb{N} \to \mathbb{N}$. Then

$$TIME[f(n)] = \{A \mid A = \mathcal{L}(M) \text{ for some } M \text{ that runs in time } f(n)\}$$

Alan Cobham and Jack Edmonds identified the complexity class, P, of problems recognizable in some polynomial amount of time, as being an excellent mathematical wrapper of the class of feasible problems – those problems all of whose moderately-sized instances can be feasibly recognized,

$$P = \bigcup_{i=1,2,\ldots} TIME[\mathcal{O}(n^i)]$$

Any problem not in P is certainly not feasible. On the other hand, natural problems that have algorithms in P, tend to eventually have algorithms discovered for them that are actually feasible.

Many important complexity classes besides P have been defined and studied; a few of these are NP, $PSPACE$, and EXP. $PSPACE$ consists of those problems solvable using some polynomial amount of memory space. EXP is the set of problems solvable in time $2^{p(n)}$ for some polynomial, p.

Perhaps the most interesting of the above classes is NP: nondeterministic polynomial time. The definition comes not from a real machine, but rather a mathematical abstraction. A nondeterministic Turing machine, N, makes a choice (guess) of one of two possible actions at each step. If, on input ω, some sequence of these choices leads to acceptance, then we say that N accepts ω, and we say the nondeterministic time taken by N on input ω, is just the number of steps taken in the sequence that leads to acceptance. A nondeterministic machine is not charged for all the other possible choices it might have made, just the single sequence of correct choices.

NP is sometimes described as the set of problems, S, that have short proofs of membership. For example, suppose we are given a list of m large natural numbers: $a_1, \ldots, a_m$, and a target number, t. This is an instance of the Subset Sum problem: is there a subset of the m numbers whose sum is exactly t? This problem is easy to solve in nondeterministic linear time: for each i, we guess whether or not to take a_i. Next we add up all the numbers we decided to take and if the sum is equal to t then accept. Thus the nondeterministic time is linear, *i.e.*, some constant times the length of the input, n. However there is no known (deterministic) way to solve this problem in time less than exponential in n.

There has been a large study of algorithms and the complexity of many important problems is well understood. In particular reductions between problems have been defined and used to compare the relative difficulty of two problems. Intuitively, we say that A is reducible to B ($A \leq B$) if there is a simple transformation, τ, that maps instances of A to instances of B in a way that preserves membership, *i.e.*, $\tau(\omega) \in B \iff \omega \in A$.

Remarkably, a high percentage of naturally occurring computational problems turn out to be complete for one of the above classes. (A problem, A, is complete for a complexity class C if A is a member of C and all other problems B in C are no harder than A, *i.e.*, $B \leq A$. Two complete problems for the same class have equivalent complexity.)

The reason for this completeness phenomenon has not been adequately explained. One plausible explanation is that natural computational problems tend to be universal in the sense of Turing's universal machine. A universal problem in a certain complexity class can simulate any other problem in that class. The reason that the class NP is so well studied is that a large number of important practical problems are NP complete, including Subset Sum. None of these problems is known to have an algorithm that is faster than exponential time, although some NP-complete problems admit feasible approximations to their solutions.

A great deal remains open about computational complexity. We know that strictly more of a particular computational resource lets us solve strictly harder problems, *e.g.*. $TIME[n]$ is strictly contained in $TIME[n^{1.01}]$ and similarly for $SPACE$ and other measures. However, the trade-offs between different computational resources is still quite poorly understood. It is obvious that P is contained in NP. Furthermore, NP is contained in $PSPACE$ because in $PSPACE$ we can systematically try every single branch of an NP computation, reusing space for the successive branches, and accepting if any of these branches lead to acceptance. $PSPACE$ is contained in EXP because if a $PSPACE$

machine takes more than exponential time, then it has exactly repeated some configuration so it must be in an infinite loop. The following are the known relationships between the above classes:

$$P \subseteq NP \subseteq PSPACE \subseteq EXP$$

However, while it seems clear that P is strictly contained in NP, that NP is strictly contained in $PSPACE$, and that $PSPACE$ is strictly contained in EXP, none of these inequalities has been proved. In fact, it is not even known that P is different from $PSPACE$, nor that NP is different from EXP. The only known proper inclusion from the above is that P is strictly contained in EXP. The remaining questions concerning the relative power of different computational resources are fundamental unsolved problems in the theory of computation.